Introdução à linguagem

SQL

Prof. Edberto Ferneda

1
Banco de Dados

Banco de Dados é um conjunto de informações organizadas que podem estar em um sistema manual ou em um sistema computadorizado.

Em um sistema manual, as informações são armazenadas em arquivos, dentro de gavetas, e a recuperação e consulta destas informações é bastante trabalhosa, pois exige uma pesquisa manual.

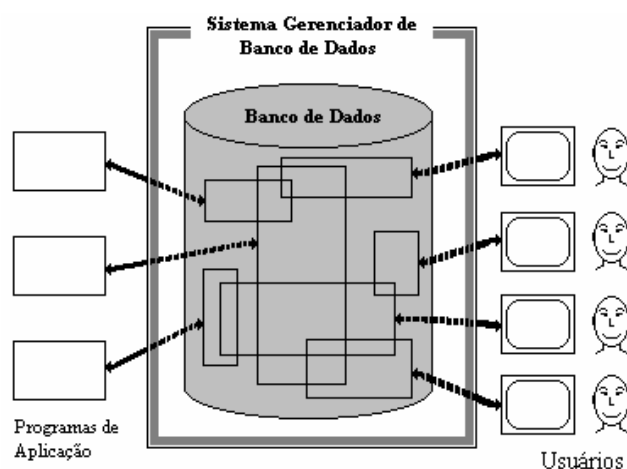
Em um sistema de computador, as informações são armazenadas em meios magnéticos, e a recuperação das informações é feita através de softwares específicos.

Vantagens no uso de computador:

- Recuperação e atualização rápida das informações;
- Informação ocupa menos espaço (meios magnéticos);
- Vários usuários podem compartilhar as informações;
- *Inexistência* dados redundantes;
- *Inexistência* de valores inconsistentes;
- Definição de regras de segurança no acesso aos dados

Sistema Gerenciador de Banco de Dados

Um sistema de gerenciamento de banco de dados (SGBD) consiste de uma coleção de dados inter-relacionados e um conjunto de programas (software) para acessar esses dados. A coleção de dados é comumente chamada de banco de dados. O principal objetivo de um SGBD é proporcionar um ambiente conveniente e eficiente para recuperar e armazenar informações no banco de dados.

Os SGBDs são concebidos para gerenciar grandes quantidades de informação. O gerenciamento dos dados envolve tanto a definição de estruturas para armazenamento das informações como a implementação de mecanismos para a manipulação dessas informações.

Além disso, um SGBD deve proporcionar a segurança das informações armazenadas no banco de dados, mesmo em caso de queda no sistema ou de tentativas de acessos não autorizados.

Os dados em um SGBD podem ser compartilhados entre diversos usuários. Para isso, um SGBD deve possuir formas de compartilhamento do banco de dados.

Devido à importância da informação na maioria das organizações, o banco de dados é um recurso valioso. Isso tem levado ao desenvolvimento de uma larga gama de conceitos e técnicas para o gerenciamento eficiente dos dados.

Componentes de um SGBD

Basicamente, um SGBD nada mais é do que um sistema de armazenamento de dados baseado em computador; isto é, um sistema cujo objetivo global é registrar e manter informações.

Um SGBD é composto de quatro componentes básicos: hardware, dados, software e usuários.

Hardware

Consiste dos meios de armazenamentos de dados – discos, fitas, etc. – nos quais reside o banco de dados, juntamente com os dispositivos associados a esses meios.

Dados

Os dados armazenados no sistema são repartidos em um ou mais banco de dados. Um banco de dados é um depósito de

dados armazenados. Geralmente ele é integrado e compartilhado.

Por “integrado” quer-se dizer que o banco de dados pode ser imaginado como sendo a unificação de diversos arquivos, eliminando total ou parcialmente qualquer redundância entre estes arquivos.

Por “compartilhado” quer-se dizer que partes individuais dos dados podem ser acessadas por diversos usuários diferentes. O compartilhamento é na realidade uma conseqüência do banco de dados ser integrado. O termo “compartilhado” é freqüentemente expandido para cobrir também o compartilhamento concorrente; isto é, a capacidade de que diversos usuários diferentes estejam tendo acesso ao banco de dados ao mesmo tempo.

Software

Entre o banco de dados físico (isto é, os dados armazenados) e os usuários do sistema encontra-se uma camada de software que é propriamente o sistema de gerenciamento de banco de dados. Todas as solicitações dos usuários para acessar o banco de dados são manipulados pelo SGBD.

Uma função geral provida pelo SGBD é isolar os usuário do banco de dados dos níveis de detalhes de hardware. Em outras palavras, o SGBD fornece uma visão do banco de dados acima do nível de hardware.

Usuários

Pode-se identificar três classes de usuários. Primeiramente temos o programador de aplicações, responsável por escrever programas de aplicação que utilizam o banco de dados. Estes programadores operam com os dados de todas as formas usuais: recuperando informações, criando novas informações, retirando ou alterando informações existentes.

A segunda classe de usuários é o usuário final, que tem acesso ao banco de dados a partir de um terminal. Um usuário final pode utilizar uma linguagem de consulta fornecida como parte integrante do sistema (SQL, por exemplo), ou pode executar uma aplicação escrita pelo programador de aplicações.

A terceira classe de usuário é o administrador de banco de dados (DBA). É parte do trabalho do DBA decidir exatamente quais informações devem ser mantidas. Deve identificar as entidades que interessam à empresa e as informações a serem registradas sobre essas entidades.

É função do DBA servir como elemento de ligação com os usuários, para garantir a disponibilidade dos dados que eles necessitam. Ele é responsável também pela organização e desempenho do sistema tendo em vista “o melhor para a empresa”.

Por que Banco de Dados?

Um Banco de Dados proporciona à empresa um controle centralizado de seus dados operacionais, um de seus ativos mais valiosos.

Dentre as diversas vantagens de um SGBD, pode-se destacar as seguintes:

- Eliminação de redundâncias de dados armazenados;
- Evitar inconsistência de dados;
- Os dados podem ser compartilhados;
- Fácil utilização de padrões;
- Restrições de segurança;
- Manutenção da integridade dos dados.

Independência de Dados

Dizemos que uma aplicação é dependente de dados quando for impossível mudar a estrutura de armazenamento ou a estratégia de acesso sem afetar a aplicação.

A independência de dados é um objetivo maior dos sistemas de banco de dados. Podemos definir independência de dados como a imunidade das aplicações a mudanças na estrutura de armazenamento ou na estratégia de acesso.

Independência física

É a capacidade de modificar o esquema físico sem afetar os componentes do Banco de Dados. Por exemplo, criar um novo índice em uma tabela.

Independência lógica

É a capacidade de modificar o esquema conceitual sem necessidade de reescrever os programas aplicativos. Por exemplo, criar um novo atributo em uma tabela.

Níveis de Abstração

Sistema gerenciador de banco de dados é uma coleção de arquivos inter-relacionados e um conjunto de programas que permitem a diversos usuários acessar e modificar esses arquivos. Um propósito central de um sistema de banco de dados é proporcionar aos usuários uma visão abstrata dos dados. Isto é, o sistema esconde certos detalhes de como os dados são armazenados ou mantidos. Porém, para que o sistema seja utilizável, os dados precisam ser recuperados eficientemente.

A preocupação com a eficiência leva a concepção de estruturas de dados complexas para a representação dos dados no banco de dados. Porém, uma vez que sistemas de banco de dados são freqüentemente usados por pessoal sem treinamento na área de informática, esta complexidade precisa ser escondida dos usuários do sistema. Isto é conseguido definindo-se diversos níveis de abstração pelos quais o banco de dados pode ser visto.

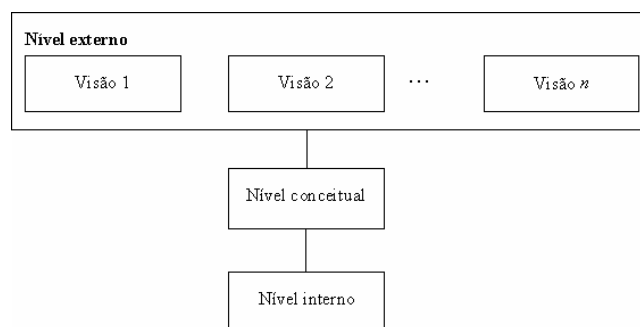
Nível físico ou interno

Este é o nível mais baixo de abstração, no qual se descreve como os dados são armazenados. Neste nível, estruturas complexas, de baixo nível, são descritas em detalhe.

Nível conceitual

Este nível é onde se descreve quais dados são armazenados e quais os relacionamentos existentes entre eles. Este nível descreve o banco de dados como um pequeno número de estruturas relativamente simples. Muito embora a implementação de estruturas simples possa envolver estruturas complexas no nível físico, o usuário do nível conceitual não necessita estar ciente disso. O nível conceitual é usado pelos administradores do banco de dados, que devem decidir qual informação deve ser mantida no banco de dados.

Nível externo

Este é o nível mais alto de abstração, no qual se expõe apenas parte do banco de dados. Apesar do nível conceitual utilizar estruturas mais simples, há ainda um tipo de complexidade resultante do grande tamanho do banco de dados. Muitos dos usuários do sistema de banco de dados não estarão preocupados com todas as informações armazenadas. Pelo contrário, os usuários necessitam apenas de uma parte do banco de dados para realizarem seu trabalho. Para simplificar a interação desses usuários com o sistema existe o nível externo, também chamado nível de visão. Pode haver diferentes visões para um mesmo banco de dados.

Linguagem de Definição de Dados

Um esquema de banco de dados é especificado por um conjunto de definições que são expressas em uma linguagem especial chamada linguagem de definição de dados (DDL). O resultado da execução de instruções DDL é um conjunto de tabelas que são armazenadas num arquivo especial chamado dicionário de dados (ou diretório de dados).

Um diretório de dados é um arquivo que contém metadados; isto é, “dados acerca dos dados”. Este arquivo é consultado antes que os dados reais sejam lidos ou modificados no sistema de banco de dados.

Linguagem de Manipulação de Dados

Entenda-se por manipulação de dados:

- A recuperação de informação armazenada no banco de dados;
- A inserção de novas informações no banco de dados;
- A remoção de informações do banco de dados

No nível físico, precisamos definir algoritmos que permitam o acesso aos dados de forma eficiente. Em níveis mais altos de abstração a ênfase está na facilidade de uso. O objetivo principal é proporcionar uma eficiente interação humana com o sistema.

Uma linguagem de manipulação de dados (DML) é uma linguagem que permite aos usuários acessar ou manipular dados organizados por um modelo de dados apropriado.

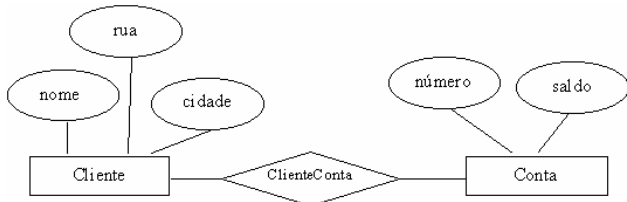
Modelo de Dados

Modelo de dados é uma coleção de ferramentas conceituais para descrição dos dados, relacionamentos entre os dados, semântica e restrição dos dados. Diversos modelos de dados foram propostos, e estão divididos em três grupos:

- Modelos baseados em objetos;
- Modelos baseados em registros;
- Modelos físicos.

Os modelos que iremos nos concentrar são os modelos baseados em registros.

Modelos Baseados em Registros

Modelos lógicos baseados em registros são usados na descrição de dados nos níveis conceitual e externo (visão). Esses modelos são usados para especificar tanto a estrutura lógica global do banco de dados como uma descrição em alto nível de implementação.

Modelo Relacional

No modelo relacional os dados e os relacionamentos entre os dados são representados por uma coleção de tabelas, cada qual com um número de colunas. Para ilustrar isto, considere um banco de dados composto de clientes e contas.

Cliente

Cod_Cli	Nome	Rua	Cidade
01	José	Pio XI	São Paulo
02	Maria	São Francisco	Recife
03	Gabriela	do Sol	Maceió

Cliente_conta

Cód_cli	Cód_cc
01	900
02	556
02	647
03	647
03	801

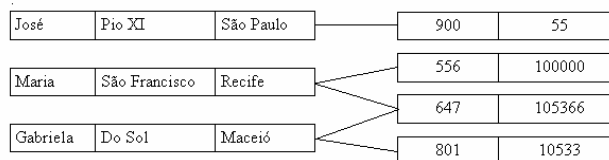
Conta

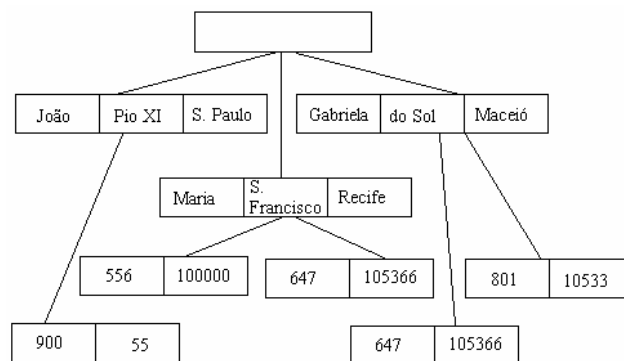
Cod_CC	Saldo
900	55,00
556	100000,00
647	105366,00
801	10533,00

Modelo de rede

Os dados no modelo de rede são representados por coleções de registros e os relacionamentos entre os dados são representados por ligações que podem ser vistas como apontadores. Os registros no banco de dados são organizados como coleções de grafos.

José	Pio XI	São Paulo
Maria	São Francisco	Recife
Gabriela	Do Sol	Maceió

900	55
556	100000
647	105366
801	10533

Modelo Hierárquico

O modelo hierárquico é similar ao modelo de rede no sentido em que dados e relacionamento são representados por registros e ligações, respectivamente. O modelo hierárquico difere do modelo em rede porque os registros são organizados como coleções de árvores em vez de grafos.

Banco de Dados Cliente/Servidor

Na arquitetura Cliente/Servidor o banco de dados fica residente em um computador chamado servidor e suas informações são compartilhadas por diversos usuários que executam aplicações em seus computadores locais (clientes). Essa arquitetura propicia uma maior integridade dos dados, pois todos os usuários estarão trabalhando com a mesma informação. A arquitetura Cliente/Servidor reduz

consideravelmente o tráfego de rede, pois retorna ao usuário apenas os dados solicitados. Por exemplo, uma base de dados com cem mil registros, se for feita uma pesquisa que encontre apenas três registros, somente esses três registros serão enviados pela rede para a máquina cliente.

Bancos de Dados Distribuídos

Um banco de dados distribuído é aquele que não é inteiramente armazenado em uma única localização física, estando disperso através de uma rede de computadores geograficamente afastados e conectados por elos de comunicação. Como um exemplo bastante simplificado, consideremos o sistema de um banco no qual o banco de dados das contas dos clientes esteja distribuído pelas agências desse banco, de tal forma que cada registro individual de conta de cliente se encontre armazenado na agência local do cliente. Em outras palavras, o dado esteja armazenado no local no qual é mais freqüentemente usado, mas ainda assim disponível (via rede de comunicação) aos usuários de outros locais. As vantagens dessa distribuição são claras: combinam a eficiência do processamento local (sem sobrecarga de comunicações) na maioria das operações, com todas as vantagens inerentes aos bancos de dados. Mas, naturalmente, também há desvantagens: podem ocorrer sobrecargas de comunicação, além de dificuldades técnicas significativas para se implementar esse sistema.

O objetivo principal em um sistema distribuído é o de que ele pareça ser, ao usuário, um sistema centralizado. Isto é, normalmente o usuário não precisará saber onde se encontra fisicamente armazenada determinada porção dos dados. Portanto, o fato de ser o banco de dados distribuído só deve ser relevante ao nível interno, e não aos níveis externo e conceitual.

2
O Modelo Relacional

Os primeiros sistemas de banco de dados se baseavam no modelo hierárquico ou no modelo em rede. Em junho de 1970 o Dr. E.F. Codd escreveu no artigo "Um Modelo Relacional de Dados para banco de dados compartilhados" o que foi considerado o primeiro projeto de um modelo relacional para sistema de banco de dados.

O modelo de dados relacional representa o banco de dados como uma coleção de tabelas. Muito embora "tabelas" envolvam noções simples e intuitivas, há uma correspondência direta entre o conceito de tabela e o conceito matemático de relação.

Nos anos seguintes à introdução do modelo relacional, uma teoria substancial foi desenvolvida para os bancos de dados relacionais. Esta teoria auxilia na concepção de banco de dados relacionais e no processamento eficiente das requisições de informação feitas pelos usuários do banco de dados.

Relação

Como o próprio nome diz, uma relação é a "matéria prima" para a construção de toda a teoria do modelo relacional e, por conseqüência, é o alicerce teórico de todo sistema de banco de dados baseado no modelo relacional.

Nos sistema de banco de dados relacionais as relações são representadas através de tabelas. Uma tabela é geralmente uma entidade identificada no processo de análise do sistema que se está implementando.

Uma tabela é constituída de linhas e colunas. Toda tabela deve possuir um nome e um conjunto de atributos (ou campos). As colunas que representam os atributos da tabela devem também possuir um nome, juntamente com o tipo de dado que será armazenado na coluna.

Cada conjunto de atributos forma uma linha (ou registro) que pode ser chamado também de tupla.

Cliente ⇒ Relação ou tabela

Código	Nome	Endereco	
123	João	Rua Pio XI	⇒ Atributo, coluna ou campo (refere-se à linha de cabeçalho)
567	Maria	Rua S. Francisco	
678	Joana	Av. Liberdade	⇒ Tupla, linha ou registros
876	Gabriela	Av. Jatiúca	
976	Ana Júlia	Av. São Paulo	

O conjunto formado pelos atributos de uma relação é também chamado de Domínio.

Álgebra Relacional

A álgebra relacional é um conjunto de operações realizadas sobre relações. Cada operação usa uma ou mais relações como seus operandos, e produz outra relação como resultado.

As operações tradicionalmente usadas na teoria dos conjuntos (**união**, **interseção**, **diferença** e **produto cartesiano**) podem também ser definidas em termos de relação. Em todas, com exceção do produto cartesiano, as duas relações do operando têm que ser união-compatíveis, isto é, elas devem possuir a mesma estrutura.

A **união** de duas relações A e B é o conjunto de todas as tuplas que pertencem a A ou B.

A **interseção** de duas relações A e B é o conjunto de todas as tuplas que pertencem a A e B.

A **diferença** entre duas relações A e B (nessa ordem) é o conjunto de todas as tuplas que pertencem a A mas não a B.

O **produto cartesiano** de duas relações A e B é o conjunto de todas as tuplas ***t*** tais que ***t*** é a concatenação de uma tupla ***a*** de A com uma tupla ***b*** pertencente a B.

União

A B ∪ A B C = A B A B C

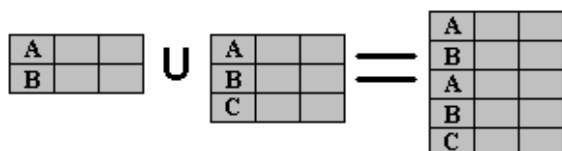
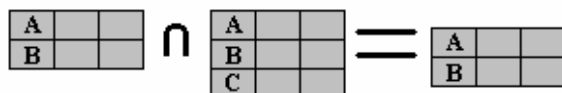
Interseção

A B ∩ A B C = A B

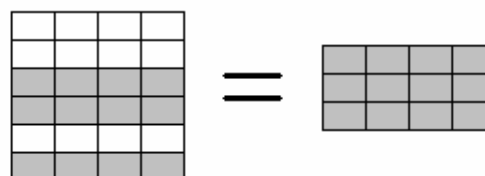
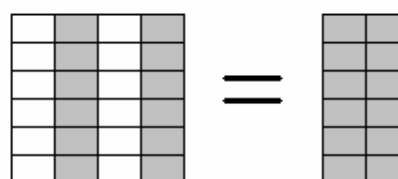
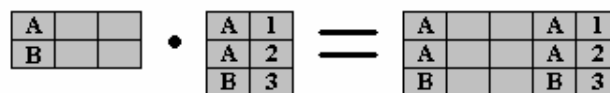
Diferença

A B C − A B = C

Produto Cartesiano

A B × 1 2 = A 1, A 2, B 1, B 2

O operador algébrico de **seleção** produz um subconjunto "horizontal" de uma dada relação. Isto é, o subconjunto de tuplas (linhas) dentro da relação dada que satisfaz a uma condição especificada.

O operador **projeção** produz um subconjunto "vertical" de uma dada relação. Isto é, o subconjunto obtido pela seleção de atributos (colunas) especificados.

3
A Linguagem SQL

Embora se fale que a linguagem SQL é uma linguagem de consulta, essa linguagem possui outras capacidades além de realizar consultas em um banco de dados. A linguagem SQL possui recursos para definição da estrutura de dados, para modificar dados no banco de dados e recursos para especificar restrições de segurança e integridade.

A versão original da linguagem SQL foi desenvolvida no laboratório de pesquisa da IBM. Esta linguagem, originalmente chamada SEQUEL, foi implementada como parte do projeto System R no início dos anos 70. A linguagem SEQUEL evoluiu e seu nome foi mudado para SQL (*Strutured Query Language*). A SQL estabeleceu-se como a linguagem padrão de consultas a banco de dados relacional. Vários Sistemas Gerenciadores de Banco de Dados suportam a linguagem SQL.. Embora existam diversas versões, com algumas diferenças entre elas, a estrutura da SQL se mantém inalterada desde a sua criação.

Um comitê foi criado para padronizar a linguagem na tentativa de torna-la independente de plataforma. O padrão SQL é definido pelo ANSI (*Amarican National Standards Institute*).

As partes da linguagem SQL

A linguagem SQL pode ser dividida em diversas partes. Algumas dessas partes serão apresentadas a seguir.

Data Definition Language (DDL)

A SQL DDL fornece comandos para definição e modificação de esquemas de relação, remoção de relações e criação de índices. Os principais comandos que fazem parte da DDL são: CREATE, ALTER, DROP.

Data Manipulation Language (DML)

A SQL DML inclui uma linguagem de consulta baseada na álgebra relacional e no cálculo relacional. Compreende também comandos para inserir, remover e modificar informações em um banco de dados. Os comandos básicos da DML são: SELECT, INSERT, UPDATE, DELETE.

Data Control Language (DCL)

É o conjunto de comandos que fazem o cadastramento de usuários e determina seu nível de privilégio para os objetos do banco de dados. Os principais comandos são: GRANT, REVOKE.

Transactions control.

A SQL inclui comandos para especificação do início e fim das transações. Diversas implementações permitem o trancamento explícito de dados para o controle de concorrência. (COMMIT, ROLLBACK, SAVEPOINT)

Tipos de Dados

Os tipos de dados existentes na linguagem SQL variam de acordo com a versão e fabricante. A primeira versão, surgida por volta de 1970, não possuía tipos de dados para armazenamento de informações multimídia como som, imagem, vídeo; tão comuns nos dias de hoje. A maioria dos Sistemas Gerenciadoras de Banco de Dados incorporou esses novos tipos de dados às suas versões da linguagem SQL. Os tipos apresentados abaixo fazem parte do conjunto de tipos do padrão ANSI 92 da linguagem SQL.

CHAR(n)	Armazena caracteres alfanuméricos de tamanho fixo ***n***.
VARCHAR(n)	Cadeia de caracteres de comprimento variável e tamanho máximo de ***n*** caracteres.
INTEGER	Dado numérico inteiro de tamanho fixo.
DECIMAL(n, m) NUMERIC(n,m)	Dado numérico de tamanho variável, sendo ***n*** o número total de dígitos e ***m*** o número de casas decimais.
BIT(n)	Seqüência ***n*** de bits.
TIME	Hora de tamanho fixo.
DATE	Data de tamanho fixo.

4
A Linguagem de Definição de Dados

Departamento

Código	Decimal(5)
Nome	Char(20)

Funcionario

Matricula	Decimal(5)
Nome	Char(30)
RG	Decimal(9)
Sexo	Char(1)
Depto	Decimal(5)
Endereco	Varchar(50)
Cidade	Char(20)
Salário	Decimal(10,2)

A Linguagem de Definição de Dados (DDL) é um conjunto específico de instruções SQL que fornece meios para a criação, alteração e exclusão de tabelas e índices.

Criando tabelas

Uma tabela é definida usando o comando **CREATE TABLE**.

```
create table T (A1 D1, A2 D2,...)
```

onde ***T*** é o nome da tabela, ***Ai*** é o nome do campo da tabela *T* e ***Di*** é o tipo do campo ***Ai***.

```
create table departamento
(Codigo     decimal(5),
 Nome       char(20) )
```

```
create table funcionario
(matricula  decimal(5),
 nome       char(30),
 rg         decimal(9),
 sexo       char(1),
 depto      decimal(5),
 endereco   varchar(40),
 cidade     varchar(20),
 salário    decimal(10,2) )
```

Uma tabela é criada inicialmente vazia, sem registros. O comando INSERT, que será visto posteriormente, é usado para carregar os dados para a relação.

Restrições de Integridade

As restrições de integridade servem para garantir as regras inerentes ao sistema que está sendo implementado, prevenindo a entrada de informações inválidas pelos usuários desse sistema. Para isso, o Sistema de Banco de Dados deve possibilitar a definição de regras de integridade a fim de evitar a inconsistência dos dados que nele serão armazenados.

Chave Primária

A função da chave primária é identificar univocamente cada registro da tabela. Toda tabela deve possuir uma chave primária, que deve ser composta por um ou mais campos.

```
create table departamento
(Codigo     decimal(5) NOT NULL PRIMARY KEY,
 Nome       char(20) )
```

```
create table funcionario
(matricula  decimal(5) NOT NULL PRIMARY KEY,
 nome       char(30),
 rg         decimal(9),
 sexo       char(1),
 depto      decimal(5),
 endereço   varchar(40),
 cidade     varchar(20),
 salário    decimal(10,2) )
```

Observação:
No Interbase é obrigatória a utilização da cláusula NOT NULL *para o(s) campo(s) da chave primária.*

Opcionalmente pode-se definir a chave primária após a especificação de todos os atributos da tabela.

```
create table funcionario
(matricula  decimal(5) NOT NULL,
 nome       char(30),
 rg         decimal(9),
 sexo       char(1),
 depto      decimal(5),
 endereço   varchar(40),
 cidade     varchar(20),
 salario    decimal(10,2),
 PRIMARY KEY (matricula) )
```

Quando uma tabela possui uma chave primaria composta por mais de um campo esta forma é obrigatória.

Evitando valores nulos

É muito comum definirmos campos que não podem conter valores nulos. Isto é, o seu preenchimento do campo é obrigatório para que se mantenha a integridade dos dados no sistema.

Para evitar que em algum momento um campo de uma tabela possa conter valor nulo (*null*) deve-se utilizar a cláusula NOT NULL após a definição do campo.

```
create table funcionario
(matricula decimal(5) NOT NULL PRIMARY KEY,
 nome       char(30) NOT NULL,
 rg         decimal(9),
 sexo       char(1),
 depto      decimal(5),
 endereço   varchar(40),
 cidade     varchar(20),
 salario    decimal(10,2) )
```

No exemplo acima, o preenchimento do campo ***nome*** será obrigatório. Caso o usuário se esqueça de preenche-lo, o SGBD apresentará uma mensagem de erro.

Evitando valores duplicados

Podem existir situações onde o valor armazenado em um campo de um registro deve ser único em relação a todos os registros da tabela. Isto é, não pode haver dois registros com o mesmo valor para um determinado campo.

Para implementar esta restrição de integridade deve-se utilizar a cláusula UNIQUE após a especificação de uma coluna.

```
create table funcionario
(matricula  decimal(5) NOT NULL PRIMARY KEY,
 nome       char(30) NOT NULL,
 rg         decimal(9) NOT NULL UNIQUE,
 sexo       char(1),
 depto      decimal(5),
 endereço   varchar(40),
 cidade     varchar(20),
 salario    decimal(10,2) )
```

No exemplo acima, caso o usuário atribua ao campo RG um valor já existente em outro registro desta mesma tabela, o SGBD apresentará uma mensagem de erro.

Observação:
No Interbase é obrigatória a utilização da cláusula NOT NULL *juntamente com a cláusula* UNIQUE.

Definindo valores default

Pode-se definir um valor padrão para um campo acrescentando à sua definição a cláusula DEFAULT. Esta cláusula permite substituir automaticamente os valores nulos por um valor inicial desejado.

```
create table funcionario
(matricula  decimal(5) NOT NULL PRIMARY KEY,
 nome       char(30) NOT NULL,
 rg         decimal(9) NOT NULL UNIQUE,
 sexo       char(1),
 depto      decimal(5),
 endereço   varchar(40),
 cidade     varchar(20) DEFAULT 'São Paulo',
 salario    decimal(10,2) )
```

Evitando valores inválidos

Existem situações onde um campo pode receber apenas alguns determinados valores. Para que o valor de um campo fique restrito a um determinado conjunto de valores, utiliza-se a cláusula CHECK.

```
create table funcionario
(matricula  decimal(5) NOT NULL PRIMARY KEY,
 nome       char(30) NOT NULL,
 rg         decimal(9) UNIQUE,
 sexo       char(1) CHECK( sexo in ('M', 'F') ),
 depto      decimal(5),
 endereco   varchar(40),
 cidade     varchar(20) DEFAULT 'São Paulo',
 salario    decimal(10,2) CHECK(salario>350) )
```

Integridade referencial

Freqüentemente desejamos que o valor armazenado em um determinado campo de uma tabela esteja presente na chave primária de outra tabela. Este atributo é chamado *chave estrangeira* (FOREIGN KEY). Por exemplo, o campo ***depto*** da tabela *funcionario* deve conter o código de um departamento anteriormente cadastrado na tabela *departamento*. Para que essa restrição seja sempre observada, utiliza-se a cláusula REFERENCES na definição do campo *depto* da tabela *funcionario*.

O campo ***depto*** da tabela *funcionario* é portanto uma *chave estrangeira* e só será permitido armazenar valores que estejam previamente cadastrado no campo ***codigo*** da tabela *departamento*.

```
create table funcioário
(matricula  decimal(5) NOT NULL PRIMARY KEY,
 nome       char(30) NOT NULL,
 rg         decimal(9) NOT NULL UNIQUE,
 sexo       char(1) CHECK( sexo in (‘M’, ‘F’),
 depto decimal(5) REFERENCES departamento(codigo),
 endereco   varchar(40),
 cidade     varchar(20) DEFAULT ‘São Paulo’,
 salario    decimal(10,2) CHECK(salario>350) )
```

Assim como na definição da chave primária, pode-se definir a chave estrangeira após a especificação de todos os campos da tabela.

```
create table cliente
(matricula  decimal(5) NOT NULL,
 nome       char(30) NOT NULL,
 rg         decimal(9) NOT NULL UNIQUE,
 sexo       char(1) CHECK(sexo in (‘M’, ‘F’)),
 depto      decimal(5),
 endereco   varchar(40),
 cidade     varchar(20) DEFAULT ‘Sao Paulo’,
 primary key (matricula),
 FOREIGN KEY (depto) REFERENCES
                   departamento(codigo) )
```

Removendo uma tabela

Para remover uma relação de um banco de dados SQL, usa-se o comando DROP TABLE. O comando DROP TABLE remove todas as informações sobre a relação.

```
drop table T
```

onde *T* é o nome de uma tabela do banco de dados

Alterando uma Tabela

O comando ALTER TABLE é usado para adicionar, excluir ou alterar atributos em uma tabela.

Incluir campos

Para inserir um novo atributo em uma tabela é usada a cláusula **add**. O novo campo terá valor *null* para todos os registros da tabela.

```
alter table T
   add A1 D1,
   add A2 D2,
   . . .
```

onde ***T*** é o nome de uma tabela e ***A****i* ***D****i* é uma lista contendo nome do atributo (***A****i*) a ser adicionado e o tipo desse atributo (***D****i*).

Excluir campos

Para excluir colunas de uma tabela utiliza-se a cláusula **drop**.

```
alter table T
   drop A1,
   drop A2,
   . . .
```

onde *T* é o nome de uma tabela e ***Ai*** é uma lista dos atributos a serem removidos.

Para alterar o nome de um atributo de uma tabela utiliza-se a cláusula **alter...to**.

```
alter table T
   alter A1 to nA1,
   alter A2 to nA2,
   . . .
```

onde ***T*** é o nome de uma tabela, ***A*** é o nome do atributo a ter o seu nome alterado para ***nA***.

Exemplo:

```
Alter table departamento
  Alter Codigo to dep_Codigo,
  Alter Nome   to dep_Nome
```

Para alterar o tipo de um atributo utiliza-se a cláusula **alter...type**.

```
alter table T
   alter A1 type t1,
   alter A2 type t2,
   . . .
```

onde ***T*** é o nome de uma tabela, *A* é o nome do atributo a ter o seu tipo alterado para ***D***.

Exemplo:

```
Alter table departamento
  Alter nome type char(30)
```

Índices

Os índices são componentes do banco de dados destinados a agilizar o acesso aos dados. Um índice pode ser associado a uma coluna ou a uma combinação de várias colunas.

Uma vez criado um índice, todas as alterações feitas à tabela são automaticamente refletidas no índice. Pode-se criar vários índices para uma tabela.

As únicas instruções SQL que tratam de índices são CREATE INDEX e DROP INDEX.

Criando índices

A instrução CREATE INDEX exige que se defina um nome para o índice a ser criado, seguido do nome da tabela e por fim, uma lista contendo o nome dos atributos que compõem o índice.

```
create index i on T(A1, A2, ...)
```

Onde ***i*** é o nome do índice, T é o nome da tabela que se deseja indexar e ***Ai*** são os atributos de indexação.

```
create index RG_Funcionário ON funcionario(RG)
```

Existem duas razões principais para se usar índices:

- Evita dados duplicados, aumentando a garantia de integridade no banco de dados com o uso da cláusula UNIQUE;
- Aumenta a rapidez do banco de dados, criando índices que sejam convenientes às consultas mais comuns e freqüentes no banco de dados.

A cláusula UNIQUE, quando utilizada, não permite que duas linhas da tabela assumam o mesmo valor no atributo ou no conjunto de atributos indexados.

```
create UNIQUE index RG_Funcionario
                              on funcionario(RG)
```

O uso de índices pode aumentar a velocidade das consultas, porém deve-se ter cautela na criação desses índices. Não há limites em relação a quantidade de índices criados. No entanto sabe-se que eles ocupam espaço em disco e nem sempre otimizam as consultas, pois não se tem o controle sobre a maneira pela qual os dados serão acessados.

Removendo índices

Para remover índices utiliza-se a instrução DROP INDEX

```
drop index i
```

onde ***i*** é o nome do índice que se deseja excluir

5

Linguagem de Manipulação de Dados

Aluno

RA	Nome	Serie	Turma	Endereço
112121	Maria Pereira	2	A	Rua Pio XII, 23
123251	José da Silva	3	B	Rua Direita, 45
321233	Rui Barros	1	B	Rua Edson, 32
453627	Ivo Pitanga	3	A	Praça Redonda, 34

Inserção de registros

Para inserir dados em uma tabela utiliza-se o comando INSERT INTO onde são especificados os valores de cada campo do novo registro.

Suponha que desejamos inserir um aluno com os seguintes dados:

```
RA       123251
Nome        José da Silva
Serie    3
Turma    B
Endereco Rua Direita, 45
```

```
insert into aluno
values (123251, 'José da Silva', 3, 'B',
        'Rua Direita, 45')
```

No exemplo acima, os valores são especificados na ordem na qual os campos foram definidos na tabela.

Caso o usuário não se lembrar da ordem dos atributos, é permitido que os atributos sejam especificados como parte da instrução INSERT.

```
insert into aluno(nome,ra,serie,endereco,turma)
values ('José da Silva', 123251, 3,
        'Rua Direita, 45', 'B')
```

Remoção de registros

A remoção de registros de uma tabela é feita através da instrução DELETE.

```
delete from T
where P
```

onde ***P*** representa um predicado (condição) e ***T*** representa uma tabela.

"Excluir todos os alunos"

```
delete from alunos
```

"Remover todos os alunos da terceira série A"

```
delete from aluno
where serie = 3
  and turma = 'A'
```

Alteração de registros

Para alterar o valor de um campo de um determinado registro ou de registros que obedecem a determinada condição utiliza-se a instrução UPDATE.

Suponha que o aluno José da Silva (ra=123251) será transferido para a quarta série.

```
update aluno
set serie = 4
where ra = 123251
```

Suponhamos agora que todos os alunos da terceira série B serão transferidos para a quarta série.

```
update aluno
set serie = 4
where serie = 3
  and turma = 'B'
```

Consultando os dados

O principal comando da Linguagem de Manipulação de dados (DML) é o comando SELECT-FROM-WHERE.

- A cláusula **SELECT** corresponde à projeção da álgebra relacional. É usada para listar os campos desejados no resultado de uma consulta.
- A cláusula **FROM** corresponde ao produto cartesiano da álgebra relacional. Na cláusula FROM são listadas todas as tabelas a serem utilizadas na consulta.
- A cláusula **WHERE** corresponde à seleção da álgebra relacional. Consiste em um predicado (condição) envolvendo atributos das tabelas que aparecem na cláusula **FROM**.

Uma típica consulta SQL tem a forma:

```
select A1, A2, A3, ...
from   T1, T2, ...
where  P
```

onde ***Ai*** representa os atributos, *Ti* as tabelas envolvidas na consulta e ***P*** um predicado ou condição.

"Apresentar os alunos da terceira série"

```
select Ra, Nome
from aluno
where serie = 3
------------------------------------
123251 José da Silva
453627 Ivo Pitanga
------------------------------------
```

A condição (ou predicado) que segue a cláusula WHERE pode conter operadores de comparação

=	Igual	>	maior	<	Menor
<>	diferente	>=	maior ou igual	<=	menor ou igual

e os operadores booleanos AND, OR e NOT.

"Apresentar o RA e o Nome dos alunos da terceira série B"

```
select ra, nome
from   aluno
where  serie = 3
  and  turma = 'B'
------------------------------------
123251 José da Silva
------------------------------------
```

A cláusula WHERE pode ser omitida e a lista de atributos (A1, A2, ...) pode ser substituída por um asterisco (*) para selecionar todos os campos de todas as tabelas presentes na cláusula FROM.

"Apresentar todos os dados dos alunos"

```
select *
from aluno
------------------------------------
112121 Maria Pereira 2 A Rua Pio XII, 23
123251 José da Silva 3 B Rua Direita, 45
321233 Rui Barros    1 B Rua Edson, 32
453627 Ivo Pitanga   3 A Praça Redonda, 34
------------------------------------
```

"Apresentar todos os dados dos alunos da terceira série"

```
select *
from aluno
where serie = 3
------------------------------------
123251 José da Silva 3 B Rua Direita, 45
453627 Ivo Pitanga   3 A Praça Redonda, 34
------------------------------------
```

Eliminando resultados duplicados

Linguagens de consultas formais são baseadas em noções matemáticas de relação. Assim, nunca deveriam aparecer registros duplicados nos resultados das consultas. Porém, como padrão, a linguagem SQL não elimina os registros duplicados que possam aparecer no resultado de uma consulta. Todavia, é possível eliminar tais duplicações através da utilização da palavra DISTINCT após a cláusula SELECT.

```
select serie
from aluno
------------------
2
3
1
3
------------------
```

```
select DISTINCT serie
from aluno
------------------
2
3
1
------------------
```

A SQL permite o uso da palavra ALL para especificar explicitamente que não queremos que as duplicações sejam removidas.

```
select ALL serie
from aluno
------------------------------------
2
3
1
3
------------------------------------
```

Uma vez que a duplicação dos registros resultantes é o padrão, a utilização da cláusula ALL torna-se opcional.

Ordenando o resultado de uma consulta

A linguagem SQL oferece uma maneira de controlar a ordem que a resposta de uma consulta será apresentada. A cláusula ORDER BY permite ordenar o resultado de uma consulta.

```
SELECT A1, A2, ...
FROM   r1, r2, ...
WHERE  P
ORDER BY   A1 [ASC/DESC],
           A2 [ASC/DESC],
           ...
```

Onde ***Ai***, após a cláusula ORDER BY, são nomes de atributos que servirão de parâmetros para o ordenamento do resultado da consulta.

A cláusula ORDER BY permite ordenar as linhas do resultado da consulta em ordem crescentes ou decrescentes. Quanto utilizada, a cláusula ORDER BY sempre deve aparecer na última linha da consulta.

As palavras **ASC** e **DESC** determinam se a ordenação será ascendente ou descendente, respectivamente. Caso nada seja especificado, é assumida a ordenação ascendente (ASC).

```
select distinct serie
from aluno
ORDER BY serie
------------------------------------
1
2
3
------------------------------------
```

"Apresentar a série e o nome do aluno em ordem decrescente da série e em ordem crescente de nome"

```
select serie, nome
from aluno
order by serie DESC, nome ASC
```

```
3 Ivo Pitanga
3 José da Silva
2 Maria Pereira
1 Rui Barros
```

Operações de Conjunto

A SQL inclui as operações UNION, INTERSECT e MINUS.

"Apresentar o nome dos aluno que cursam a terceira série"

```
select nome
from aluno
where serie = 3
```

"Apresentar o nome dos alunos que estejam na turma B"

```
select nome
from aluno
where turma = 'B'
```

Para achar todos os alunos que cursam a terceira série ou que estejam na turma B podemos utilizar o seguinte comando:

```
  select nome, serie, turma
  from aluno
  where serie = 3
UNION
  select nome, serie, turma
  from aluno
  where turma = 'B'
```

```
José da Silva 3 B
Ivo Pitanga   3 A
Rui Barros    1 B
```

Para achar todos os alunos que cursam a terceira série **e** que são da turma B pode-se fazer:

```
  select nome, serie, turma
  from aluno
  where serie = 3
INTERSECT
  select nome, serie, turma
  from aluno
  where turma = 'B'
```

```
José da Silva 3 B
```

Para achar todos os alunos que cursam a segunda **ou** a terceira série:

```
  select nome, serie, turma
  from aluno
  where serie = 2
UNION
  select nome, serie, turma
  from aluno
  where serie = 3
```

```
Maria Pereira 2 A
José da Silva 3 B
Ivo Pitanga   3 A
```

Como padrão, a operação UNION elimina as linhas duplicadas. Para reter duplicações precisamos escrever UNION ALL no lugar de UNION.

Para encontrar todos os alunos que fazem a terceira série mas não são da turma B podemos escrever:

```
  select nome
  from aluno
  where serie = 3
minus
  select nome
  from aluno
  where turma = 'B'
```

```
Ivo Pitanga   3 A
```

As operações INTERSECT e MINUS eram parte da SQL original mas não estão incluídas na versão padrão pois é possível expressar estas operações de uma outra forma.

6

Predicados

Cliente

Codigo	Decimal(5)
Nome	char(30)
Endereco	char(40)
Cidade	char(20)

Conta

Numero	Decimal(7)
Agencia	Decimal(4)
Cliente	Decimal(5)
Saldo	decimal(16,2)

Emprestimo

Agencia	decimal(4)
Numero	decimal(7)
Cliente	decimal(5)
Valor	decimal(16,2)

Agencia

Código	decimal(4)
Cidade	char(20)
Ativos	decimal(16,2)

"Apresentar o número da(s) conta(s) e o respectivo saldo dos clientes que possuem empréstimo"

```
select conta.numero, saldo
from emprestimos, conta
where emprestimo.cliente = conta.cliente
```

Note que a SQL usa a notação ***relação.atributo*** para evitar ambigüidade nos casos em que um atributo aparece no esquema de mais de uma relação. Poderia ter sido escrito **conta.saldo** em vez de **saldo** na cláusula SELECT. No entanto, uma vez que o atributo **saldo** aparece em apenas uma das relações referenciadas na cláusula FROM, não há ambigüidade quando escrevemos apenas **saldo**.

Vamos entender a consulta anterior e considerar um caso mais complicado no qual queremos também os clientes com empréstimos na agência Ipiranga:

"Apresentar o nome e a cidade dos clientes com empréstimo na agência Ipiranga"

Para escrever esta consulta, devemos utilizar duas restrições na cláusula WHERE, conectadas pelo operador lógico AND.

```
select nome, cidade
from emprestimo, cliente
where cliente=cliente.codigo
 and  agencia = 'Ipiranga'
```

A linguagem SQL utiliza conectivos lógicos **and**, **or** e **not**. Além disso, uma expressão aritmética pode envolver qualquer um dos operadores +, -, * e /.

A SQL inclui ainda um operador de comparação **between** para simplificar as cláusulas **where** que especificam que um valor seja menor ou igual a um determinado valor e maior ou igual a um outro valor. Se desejarmos achar o número das contas com saldo entre 90.000 e 100.000, podemos usar a cláusula BETWEEN:

```
select numero
from conta
where saldo between 90000 and 100000
```

ao invés de

```
select numero
from conta
where saldo >= 90000 and saldo <= 100000
```

Da mesma forma, pode-se usar o operador de comparação **NOT BETWEEN**.

A SQL inclui um operador de substituição de cadeia para comparações em cadeias de caracteres. Os padrões são descritos usando dois caracteres especiais:

- % (por cento) Substitui qualquer subcadeia
- _ (sublinhado) Substitui qualquer caracter

Os padrões são sensíveis à forma; isto é, os caracteres maiúsculos não substituem os caracteres minúsculos, ou vice-versa. Para ilustrar a substituição de padrão, considere os seguintes exemplos:

- 'Jose%' substitui qualquer cadeia começando com "Jose";
- '%ari%' substitui qualquer cadeia contendo "ari" como uma subcadeia, por exemplo, "Maria", "Mariana", "Itaparica";
- '_ _ _' substitui qualquer cadeia com exatamente três caracteres;

- '_ _ _ %' substitui qualquer cadeia com pelo menos três caracteres.

Os padrões são expressos em SQL usando o operador de comparação LIKE. Considere a consulta:

"Apresentar os nomes de todos os clientes cujas ruas possuem a subcadeia 'Lima' "

```
select nome
from cliente
where endereco like "%Lima%"
```

Para que os padrões possam incluir os caracteres especiais % e _, a linguagem SQL permite a especificação de um caractere de escape, representado por um caracter definido pelo usuário, por exemplo "\" (barra invertida). Assim, o caracter após "\" é interpretado como um literal, não como um caracter especial. Por exemplo:

Like 'ab\%cd%' escape '\'	Cadeias de caracteres que começam com "ab%cd";
Like 'ab\\%cd%' escape '\'	Cadeias de caracteres que começam com "ab\cd".

A SQL permite a procura por não-substituição em vez de substituição usando o operador de comparação NOT LIKE.

Membros de Conjuntos

O conectivo **IN** testa os membros de conjunto, onde o conjunto é uma coleção de valores produzidos por uma cláusula SELECT. O conectivo **NOT IN** testa a ausência dos membros de um conjunto.

"Apresentar os clientes que possuem conta e empréstimo na agência 38"

Começamos localizando todos os possuidores de conta na agencia 38:

```
select cliente
from conta
where agencia = 38
```

Precisamos então encontrar aqueles clientes que são solicitadores de empréstimo da agência 38 e que aparece na lista de possuidores de contas da agência 38. Fazemos isso embutindo a subconsulta acima em outro SELECT.

```
select cliente
from emprestimo
where agencia = 38
  and cliente in
         (select cliente
          from conta
          where agencia = 38 )
```

É possível escrever a mesma consulta de diversas formas em SQL. Isto é benéfico, uma vez que permite a um usuário pensar sobre a consulta no modo que lhe aparenta ser mais natural.

No exemplo anterior, testamos membros de uma relação de um atributo. É possível testar um membro em uma relação arbitrária.

```
select cliente
from emprestimo
where agencia = 38
  and agencia, cliente in
      (select agencia, cliente
       from conta)
```

Ilustramos agora o uso de NOT numa construção.

"Apresentar todos os clientes que têm uma conta na agência 38, mas NÃO possuem um empréstimo nessa mesma agência"

```
select cliente
from conta
where agencia = 38
  and cliente not in
          (select cliente
           from emprestimo
           where agencia = 38)
```

Variáveis Tupla

Uma variável tupla na linguagem SQL precisa estar associada à uma relação particular. As variáveis tupla são definidas na cláusula FROM.

"Apresentar o nome e a cidade dos clientes que possuem empréstimo"

```
select C.nome, C.cidade
from emprestimo E, cliente C
where E.cliente = C.codigo
```

Uma variável tupla é definida na cláusula FROM depois do nome da relação à qual está associada, separada por um ou mais espaços.

Variáveis tupla definidas em uma consulta são válidas também nas subconsultas de nível inferior. Variáveis tupla

definidas em uma subconsulta não são "enxergadas" por subconsultas de nível superior. Se uma variável tupla é definida tanto localmente em uma subconsulta quanto globalmente em uma consulta, a definição local prevalece. Isto é análogo às regras usuais de alcance usadas para variáveis em linguagens de programação. Quando escrevemos expressões da forma **relação.atributo**, o nome da relação é, de fato, uma variável tupla definida implicitamente.

As variáveis tupla são mais úteis para comparação de duas tuplas na mesma relação.

"Quais os clientes que possuem conta em alguma agência onde o cliente de código 12345 tem conta"

```
select distinct C.cliente
from conta C, conta T
where C.cliente = 12345
  and C.agencia = T.agencia
```

Observe que não podemos usar a notação **contas.agencia**, uma vez que não estaria claro qual referência a contas é a desejada.

Um modo alternativo para expressar esta consulta é

```
select distinct cliente
from conta
where agencia in
      (select agencia
       from conta
       where cliente = 12345)
```

7
Comparação de Conjuntos

"Apresentar os nomes das agências que possuem ativos maior do que alguma agência localizada em São Paulo"

```
select distinct T.nome
from agencia T, agencia S
where T.ativos > S.ativos and
      S.cidade = 'Sao Paulo'
```

Uma vez que isto é uma comparação "maior que", não podemos escrever a expressão usando a construção in.

A SQL oferece um estilo alternativo para escrever a consulta acima. A frase "maior do que algum" é representado na SQL por **>some**. Esta construção permite reescrever a consulta em uma forma que se assemelha intimamente à fórmula da nossa consulta em português.

```
select nome
from agencia
where ativos >some
      (select ativos
       from agencia
       where cidade = 'Sao Paulo')
```

A subconsulta

```
      (select ativos
       from agencia
       where cidade = 'Sao Paulo')
```

gera o conjunto de todos valores de ativos das agências em São Paulo: A comparação **>some** na cláusula WHERE do SELECT externo é verdadeira se o valor do atributo ativos da tupla for maior do que pelo menos um membro do conjunto de todos os valores de ativos das agências de Sao Paulo.

A SQL também permite comparações **<some**, **<=some**, **>=some**, **=some** e **<>some**. Observe que =some é idêntico a in. Uma cláusula similar à cláusula some é a cláusula **any**, que possui também todas as variações existentes na cláusula **some**: **<any**, **<=any**, **>=any**, **>any**, **=any**, **<>any**.

"Apresentar os nomes das agências que possuem ativos maiores do que qualquer (todas) uma das agências de São Paulo"

A construção **>all** corresponde à frase "maior do que todos". Usando esta construção, escrevemos a consulta como segue:

```
select nome
from agencia
where ativos >all
      (select ativos
       from agencia
       where cidade = "Sao Paulo")
```

Como na cláusula some, a SQL permite comparações **<all**, **<=all**, **>=all**, **=all** e **<>all**.

As construções in, **>some**, **>all**, etc; nos permitem testar um valor simples contra membros de um conjunto. Uma vez que SELECT gera um conjunto de tuplas, podemos querer comparar conjuntos para determinar se um conjunto contém todos os membros de algum outro conjunto. Tais comparações são feitas na SQL usando as construções **contains** e **not contains**.

"Apresentar o código dos clientes que possuem conta em todas as agências localizadas em São Paulo".

Para cada cliente, precisamos ver se o conjunto de todas as agências na qual um cliente possui uma conta contém o conjunto de todas as agências em São Paulo.

```
select distinct S.cliente
from conta S
where (select T.agencia
       from conta T
       where S.cliente = T.cliente)
   contains
      (select agencia
       from agencia
       where cidade = 'Sao Paulo')
```

A subconsulta

```
      (select agencia
       from agencia
       where cidade = 'Sao Paulo')
```

encontra todas as agências em São Paulo.

A subconsulta

```
      (select T.agencia
       from conta T
       where S.cliente = T.cliente)
```

encontra todas as agências nas quais o cliente S.nome_cliente tem uma conta. Assim, o SELECT externo pega cada cliente e testa se o conjunto das agências onde

ele possui conta contém o conjunto de todas as agências em São Paulo.

A construção **contains** não aparece no padrão ANSI. Uma boa razão para isso é que o processamento da construção **contains** é extremamente custoso.

Testando Relações Vazias

A SQL inclui um recurso para testar se uma subconsulta tem alguma tupla em seus resultados. A construção **exists** retorna o valor ***true*** se o resultado da subconsulta não é vazio.

"Apresentar os clientes que possuem uma conta e um empréstimo na agência 17"

```
select nome
from cliente
where exists (select *
            from conta
            where conta.cliente = cliente.codigo
            and   agencia = 17)
and exists (select *
            from emprestimo
            where emprestimos.cliente =
                              clientes.código
            and   agencia = 17)
```

A primeira subconsulta **exists** testa se o cliente tem uma conta na agência Ipiranga. A segunda subconsulta **exists** testa se o cliente tem um empréstimo na agência Ipiranga.

A não-existência de tuplas em uma subconsulta pode ser testada usando a construção **not exists**.

"Apresentar os clientes que possuem conta na agência Ipiranga mas não possuem empréstimo nesta agência"

```
select nome_cliente
from clientes
where exists (select *
         from contas
         where contas.nome_cliente =
             clientes.nome_cliente and
             nome_agencia = 'Ipiranga')
   and not exists (select *
            from emprestimos
            where emprestimos.nome_cliente =
                clientes.nome_cliente and
                nome_agencia = 'Ipiranga')
```

"Apresentar os clientes que possuem conta em todas as agências localizadas em São Paulo"

Usando uma construção **minus**, podemos escrever a consulta da seguinte forma:

```
select distinct S.nome_cliente
from contas S
where not exists ( ( select nome_agencia
              from agencias
              where cidade = 'Sao Paulo')
            minus
            ( select T.agencia
              from contas T
              where S.nome_cliente =
                  T.nome_cliente ) )
```

A subconsulta

```
            ( ( select nome_agencia
              from agencias
              where cidade = 'Sao Paulo')
```

encontra todas as agências em São Paulo.

A subconsulta

```
            ( select T.agencia
              from contas T
              where S.nome_cliente =
                  T.nome_cliente ) )
```

encontra todas as agências na qual S.nome_cliente possui uma conta. Assim, o SELECT externo pega cada cliente e testa se o conjunto de todas as agências de São Paulo menos o conjunto de todas as agências nas quais o cliente tem uma conta, é vazio.

8
Funções Agregadas

Clientes

Nome_cliente	char(30)
Endereco	char(40)
Cidade	char(20)

Contas

Numero_conta	decimal(7)
Nome_agencia	char(20)
Nome_cliente	char(30)
Saldo	decimal(16,2)

Emprestimos

Nome_agencia	char(20)
Numero	decimal(7)
Nome_cliente	char(30)
Valor	decimal(16,2)

Agencias

Nome_agencia	char(20)
Cidade	char(20)
Ativos	decimal(16,2)

A SQL oferece a habilidade para computar funções em grupos de registros usando a cláusula **group by**. O atributo ou atributos utilizados na cláusula **group by** são usados para formar grupos. Registros com o mesmo valor em todos os atributos na cláusula **group by** são colocados em um grupo. A SQL inclui funções para computar:

A linguagem SQL possui algumas funções específicas para cálculos em grupos de tuplas:

- média: **avg**
- mínimo: **min**
- máximo: **max**
- total: **sum**
- contar: **count**

As operações como a **avg** são chamadas funções agregadas porque operam em agregações de tuplas. O resultado de uma função agregada é um valor único. Para ilustrar, considere a consulta:

"Apresentar o saldo médio de conta em cada agência"

```
select nome_agencia, avg(saldo)
from contas
group by agencia
```

A retenção de duplicatas é importante na computação da média. Suponha que os saldos de conta na agência Ipiranga sejam 1.000, 2.000, 3.000 e 1.000. O saldo médio é 7.000/4 = 1.666,67. Se as duplicações fossem eliminadas, teríamos uma resposta errada (6.000/3 = 2.000).

A cláusula **group by** conceitualmente rearranja a tabela especificada após a cláusula FROM em partições ou grupos, de tal forma que dentro de qualquer dos grupos todas as linhas tenham o mesmo valor do atributo especificado no **group by**.

Existem casos nos quais as duplicações precisam ser eliminadas antes de uma função agregada ser computada. Se desejarmos eliminar duplicações, usamos a palavra chave **distinct** na expressão agregada.

"Encontre o número de correntistas de cada agência"

Neste caso, um correntista é contado uma só vez, independentemente do número de contas que ele possa ter.

```
select nome_agencia,count(distinct nome_cliente)
from contas
group by agencia
```

Às vezes é útil definir uma condição que se aplique a grupos em vez de registros. Por exemplo, podemos estar interessados apenas em agências nas quais a média dos saldos é maior do que 1.200. Esta condição não se aplica a registros simples, mas sim a cada grupo construído pela cláusula **group by**. Para expressar tal consulta usamos a cláusula **having**. Os predicados na cláusula **having** são aplicados depois da formação dos grupos, para que funções agregadas possam ser usadas.

```
select nome_agencia, avg(saldo)
from contas
group by agencia
having avg(saldo) > 1200
```

As funções agregadas não podem ser compostas em SQL. Isto significa que qualquer tentativa de usar **max(avg(...))** não será permitida. Por outro lado, nossa estratégia é achar aquelas filiais para as quais a média de saldo é maior ou igual a todas as médias de saldo.

"Apresentar o nome das agências com a maior média de saldos"

```
select nome_agencia
from contas
group by agencia
having avg(saldo) >=all      (select avg(saldo)
                              from contas
                              group by agencia)
```

Às vezes desejamos tratar a relação inteira como um grupo simples. Em tais casos, não usamos a cláusula **group by**.

"Apresentar a média dos saldos"

```
select avg(saldo)
from contas
```

A função agregada **count** é usada freqüentemente para contar o número de tuplas numa relação. A notação para isto é **count(*).** Assim para achar o número de tuplas da relação cliente, escrevemos:

```
select count(*)
from clientes
```

Se uma cláusula WHERE e uma cláusula **having** aparecem em uma mesma consulta, o predicado na cláusula WHERE é aplicado primeiro. As tuplas que satisfazem o predicado WHERE são então agrupadas po uma cláusula **group by**. A cláusula **having** é então aplicada a cada grupo. Os grupos que satisfazem o predicado da cláusula **having** são usados pela cláusula SELECT para gerar tuplas do resultado da consulta. Se não houver uma cláusula **having**, todo o conjunto de tuplas que satisfazem a cláusula WHERE é tratado como um grupo simples.

"Apresentar a média dos saldos dos correntistas que vivem em São Paulo e possuem pelo menos três contas"

```
select avg(saldo)
from contas, clientes
where contas.nome_cliente =
                         clientes.nome_cliente
       and cidade = 'Sao Paulo'
group by contas.nome_cliente
having count(distinct numero_conta) >= 3
```

A versão ANSI da SQL requer que **count** seja usada apenas como **count(*)** ou **count(distinct...).** É válido usar **distinct** com **max** e **min** mesmo que o resultado não se altere. A palavra-chave **all** pode ser usada no lugar de **distinct** para permitir duplicações, mas, uma vez que **all** é o *default*, não existe necessidade de utilizá-lo.

A SQL inclui as operações da álgebra relacional fundamental. O produto cartesiano é representado pela cláusula FROM. A projeção é executada na cláusula SELECT. Os predicados de seleção da álgebra relacional são representados nas cláusulas WHERE. A álgebra relacional e a SQL incluem a união e a diferença. A SQL permite resultados intermediários para ser guardados em relações temporárias. Assim, podemos codificar qualquer expressão da álgebra relacional na SQL.

A SQL oferece uma rica coleção de recursos, abrangendo funções agregadas, ordenação de tuplas e outras capacidades não incluídas nas linguagens formais de consulta. Assim, a SQL é mais poderosa do que a álgebra relacional.

Muitas versões da SQL permitem que consultas SQL sejam submetidas a partir de um programa escrito em uma linguagem de uso genérico como Pascal, PL/I, Fortran, C ou Cobol. Esta forma da SQL estende ainda mais a habilidade do programador de manipular o banco de dados.

9
Visões

Uma visão (view) é uma tabela virtual cujo conteúdo é definido por uma consulta ao banco de dados. A visão não é uma tabela física, mas um conjunto de instruções que retorna um conjunto de dados. Uma visão pode ser composta por algumas colunas de uma única tabela ou por colunas de várias tabelas.

O uso de visões é particularmente útil quando se deseja dar foco a um determinado tipo de informação mantida pelo banco de dados. Imagine um banco de dados corporativo que é acessado por usuários de vários departamentos. As informações que a equipe de vendas manipula certamente são diferentes daquelas do departamento de faturamento. Por meio de visões é possível oferecer ao usuário apenas as informações que necessita, não importando se elas são oriundas de uma ou várias tabelas do banco de dados.

As visões permitem que diferentes usuários vejas as mesmas informações sob um ponto de vista diferente. As visões permitem que informações sejam combinadas par atender a um determinado usuário e até mesmo serem exportadas para outros aplicativos.

Uma das maiores vantagens de se criar uma visão é facilitar as consultas dos usuários que só utilizam determinadas informações, diminuindo assim o tamanho e a complexidade dos comandos SELECT.

Uma outra vantagem de utilizar visões é quanto a segurança, pois evita que usuários possam acessar dados de uma tabela que podem ser confidenciais.

Uma visão é definida na SQL usando o comando CREATE VIEW. Para definir uma visão precisamos dar à visão um nome e definir a consulta que a processa. A forma do comando CREATE VIEW é:

```
create view v as <expressão de consulta>
```

<expressão de consulta> é qualquer expressão de consulta válida. O nome da visão é definido por ***v***.

```
create view todos_clientes as
     (select nome_agencia, nome_cliente
      from contas)
   union
     (select nome_agencia, nome_cliente
      from emprestimos)
```

Nomes de visões aparecem em qualquer lugar que um nome de relação possa aparecer. Usando a visão todos_clientes, podemos achar todos os clientes da agência Ipiranga:

```
select nome_cliente
from todos_clientes
where nome_agencia = 'Ipiranga'
```

Uma vez que a SQL permite a um nome de visão aparecer em qualquer lugar em que o nome de uma relação aparece, podemos escrever:

```
create view emprestimos_info as
   select nome_agencia, numero, nome_cliente
   from emprestimos
```

```
insert into emprestimos_info
values ('Ipiranga', 17, 'Paulo Farias')
```

Esta inserção é na verdade uma inserção na relação empréstimo, uma vez que empréstimo é a relação a partir da qual a visão emprestimo_info foi construída. Devemos, entretanto, ter algum valor para quantia. Este valor é um valor nulo. Assim, o insert acima resulta na inserção da tupla

```
('Ipiranga', 17, 'Paulo Farias')
```

na relação emprestimos.

10
Permissões

Uma importante tarefa que deve ser realizada pelo administrador de banco de dados é a criação de contas de usuários. Qualquer pessoa que quiser acessar um banco de dados precisa ser previamente cadastrada como usuário do banco de dados e ter estabelecido para ela privilégios com relação às tarefas que poderão ser executadas no banco de dados.

Controlar o aceso ao banco de dados é uma das principais tarefas que um administrador tem. Para realizar esse controle, os bancos de dados contam com um mecanismo que permite cadastrar um usuário. Cada usuário cadastrado recebe uma senha de aceso que precisa ser fornecida em diversas situações.

Privilégios

Um privilégio é uma autorização para que o usuário acesse e manipule um objeto de banco de dados de uma certa forma. Por exemplo, um usuário pode ter o privilégio de selecionar tabelas, porém não pode modifica-las. Outro usuário pode tanto ler como alterar os dados ou até mesmo a estrutura das tabelas e outros objetos.

Existem dois tipos de privilégio: os privilégios de sistema e os privilégios de objetos.

Um privilégio de sistema é o direito ou permissão de executar uma ação em um tipo específico de objeto de banco de dados.

O privilégio de objeto é o direito de executar uma determinada ação em um objeto específico, como o direito de incluir um registro em uma determinada tabela. Os privilégios de objeto não se aplicam a todos os objetos de banco de dados.

Quando um usuário cria um objeto como uma tabela, ela só pode ser visualizada pelo próprio usuário que a criou. Para que outro usuário tenha acesso a ela, é necessário que o proprietário da tabela conceda privilégios para o usuário que irá acessar a tabela.

Atribuindo Privilégios

O comando GRANT permite atribuir privilégios a um usuário (ou grupo de usuários). Os privilégios podem ser: SELECT, INSERT, UPDATE, DELETE ou ALL PRIVILEGES. Os objetos que geralmente se concedem privilégios são tabelas e visões.

```
GRANT privilégio/ALL PRIVILEGES
ON objeto
TO usuário1, usuário2,... /PUBLIC
[WITH GRANT OPTION]
```

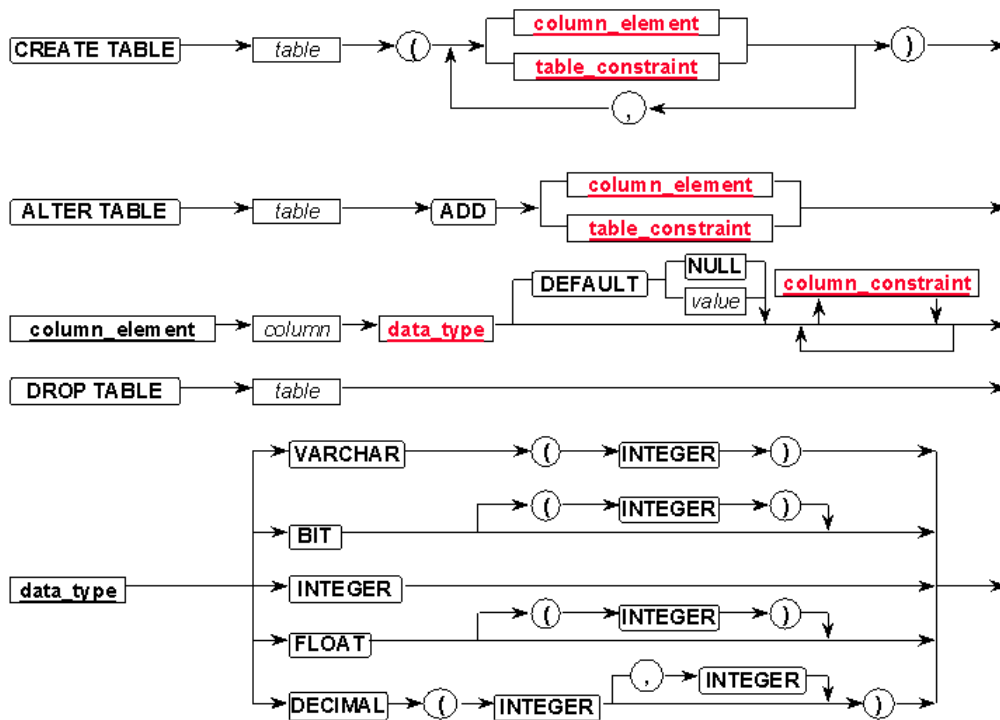
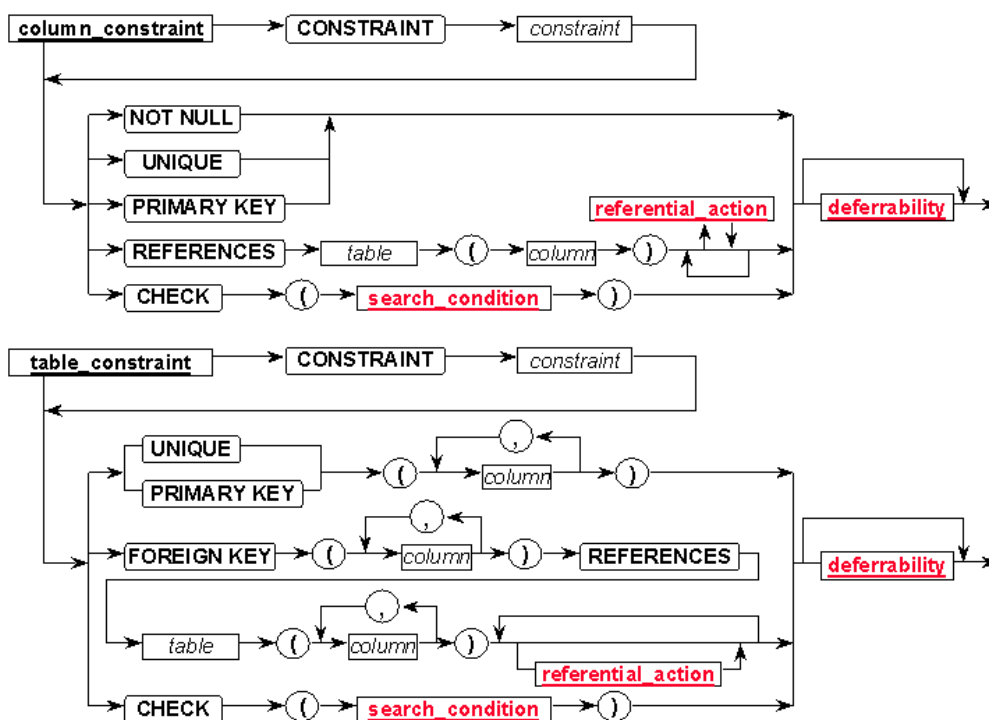
privilégio	nome do privilégio
ALL PRIVILEGES	todos os privilégios
Objeto	Geralmente tabela ou visão
Usuário	um determinado usuário
PUBLIC	todos os usuários
WITH GRANT OPTION	parâmetro opcional que permite que o usuário que recebe o privilégio possa concede-lo a outros usuários

Revogando um Privilégio

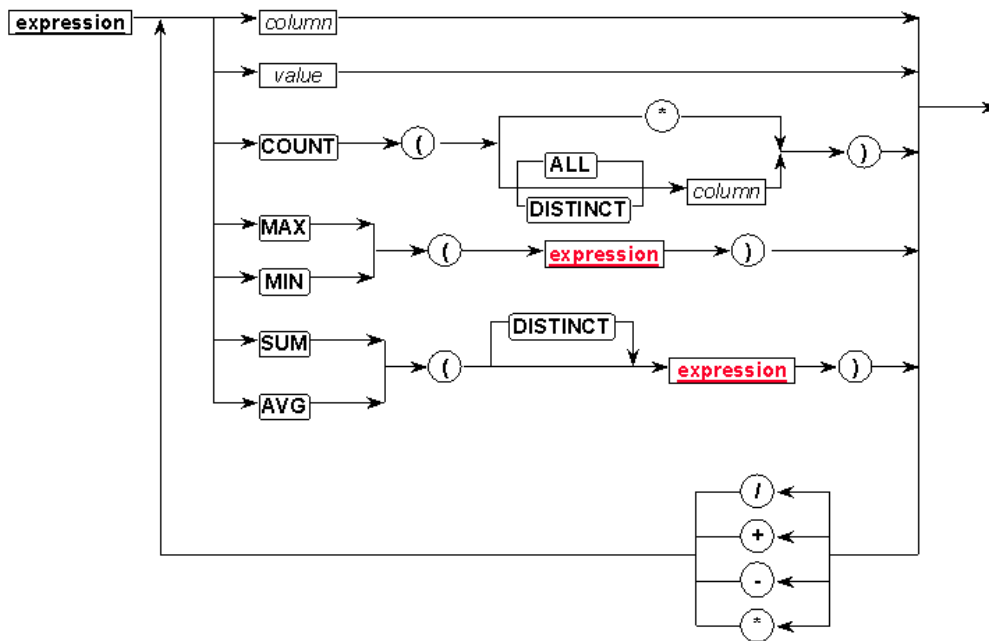
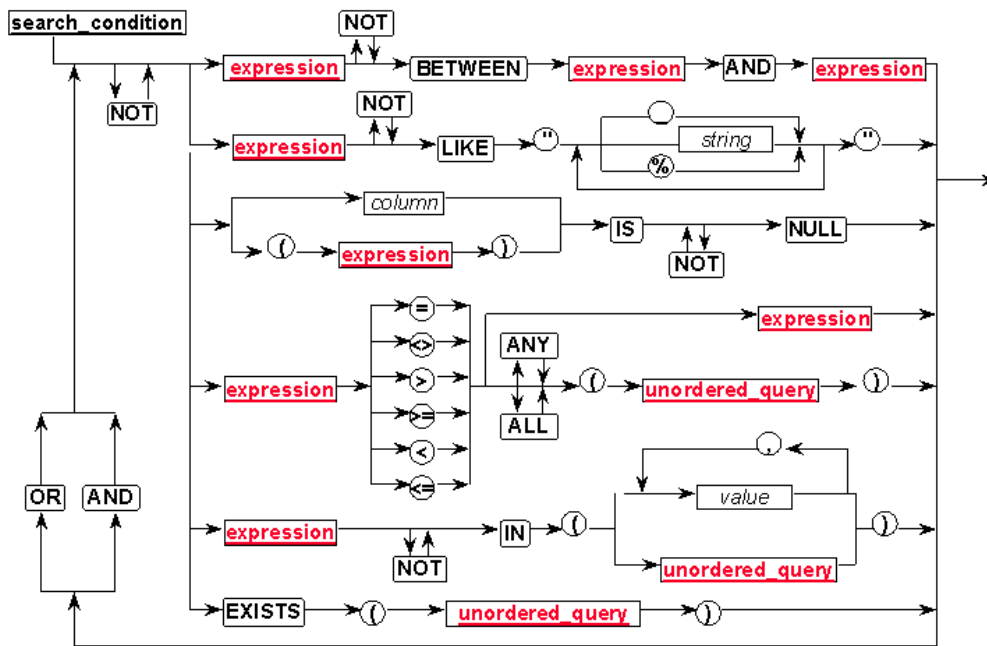
Assim como você concedeu um privilégio, também pode retirá-lo. O comando SQL responsável por essa tarefa é o comando REVOKE.

```
REVOKE [GRANT OPTION FOR] privilégio
ON objeto
FROM usuário1, usuário2,.../PUBLIC
```

GRANT OPTION FOR	parâmetro opcional que retirar a permissão de repassar o privilégio
privilégio	nome do privilégio
ALL PRIVILEGES	todos os privilégios
Objeto	Geralmente tabela ou visão
Usuário	um determinado usuário
PUBLIC	Todos os usuários

11
A Sintaxe da linguagem SQL

Assertion, Deferrablity,
Referential Action
CREATE ASSERTION
assertion
CHECK
(
search_condition
)
deferrability
deferrability
INITIALLY DEFERRED
INITIALLY IMMEDIATE
NOT
DEFERRABLE
NOT
DEFERRABLE
INITIALLY DEFERRED
INITIALLY IMMEDIATE
referential_action
ON UPDATE
ON DELETE
CASCADE
SET NULL
SET DEFAULT
NO ACTION
Query
select_block
SELECT
ALL
DISTINCT
*
expression
,
FROM
table
,
correlation_var
WHERE
search_condition
GROUP BY
column
,
HAVING
search_condition
unordered_query
select_block
UNION
INTERSECTION
EXCEPT
ordered_query
unordered_query
ORDER BY
INTEGER
column
DESC
ASC
,

Data Manipulation

Expression

Search Condition

View Access Rights

Embedded SQL
DECLARE
cursor
CURSOR FOR
ordered_query
OPEN
cursor
FETCH
cursor
INTO
:
variable
,
CLOSE
cursor
DELETE FROM
table
WHERE
CURRENT OF
cursor
UPDATE
table
SET
,
column
=
NULL
expression
(
unordered_query
)
WHERE
CURRENT OF
cursor